Knowledge Systems Laboratory
Report No. KSL 86-19

April 1986

Poligon, A System for Parallel Problem Solving

by

James P. Rice

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

To Appear in Proceedings of DARPA Workshop on Expert Systems Technology Base, Asilomar, April 1986

Table of Contents

Summary

The **Poligon**[1] system is a new, domain-independent language and attendant support environment, which has been designed specifically for the implementation of applications using a Blackboard-like problem-solving framework in a parallel computational environment.

This paper describes the Poligon system and the Poligon language, its salient and novel features. Poligon is compared with other approaches to the programming of parallel systems.

1. Introduction

The larger project of which Poligon is only a small part will not be discussed here in any detail. Design decisions made in other parts of the project will be held to be axiomatic, though some mention of these decisions will be made in order to show the motivation for the features of Poligon. The primary objective of the overall project is to achieve significant speedup of knowledge based systems, particularly those directed at real-time signal understanding.

The purpose of the Poligon language is to express the problem solving behaviour of human experts in order to map them onto a problem solving framework, which will run on simulated parallel hardware.

The fields of knowledge representation and problem solving are rich and complex. This paper will not go into any great detail in describing the problem solving processes involved. Poligon tries usefully to express knowledge both in a declarative and procedural sense, through rules [Davis 77]; and in a structural sense, through the configuration of the solution space. These will be described below.

Some crucial design criteria and early design commitments have affected the development of Poligon, the consequences of which will be described in this paper. These can be summarised as follows.

- Poligon is intended to be a language for both problem solving and the general purpose programming necessary to support it. Unlike most programs, Poligon programs must also address the problems of real-time processing, including asynchronous events and input data backup. Poligon, therefore, must assist in this respect.
- The overall project's strategy is to solve problems significantly faster than existing systems through the exploitation of parallelism. Poligon is targeted at a MIMD, distributed-memory, message-passing machine with ~thousands of processors. This hardware gives direct support for futures, remote objects and such efficient message-passing strategies as *Broadcast* and *Multicast* so as to take full advantage of its processor interconnection network.
- A consequence of the desire to achieve a significant order of parallelism in Poligon programs is that many of the control mechanisms used in serial problem solving systems, such as schedulers and event queues, have been discarded because they are highly serial. Most actions in Poligon programs are, therefore, performed asynchronously. Rules, the primary mechanism in Poligon for describing things and for getting things done, are activated as daemons. Much of the work in Poligon is aimed at providing mechanisms to cope with this chaotic behaviour.

This paper contains the following;

---

[1]The author gratefully acknowledges the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

- A discussion of related work in parallel languages.
- A discussion of the design approach guiding the development of Poligon.
- A description of the abstraction mechanisms provided by the Poligon system with some small examples.
- Some concluding remarks.
- References for further reading on the subject.

1.1. Knowledge Representation and Problem Solving in Poligon

The primary purpose of this paper is to discuss the Poligon language. It is, however, not possible completely to divorce this from the underlying hardware and from its purpose; knowledge representation and problem solving.

Poligon can be described loosely as a "Blackboard System". What this means in practice is that the problem solving metaphor of Poligon is one of cooperating experts gathered around a blackboard, posting ideas about their deductions on the blackboard. For an exposition on the term "Blackboard System" the reader is encouraged to read [Nii 86]. Poligon tries usefully to express knowledge both in a declarative and procedural sense, through rules and functions; and in a structural sense, through the configuration of the solution space on the blackboard. In particular, the term "blackboard" will be used to describe the set of all of the nodes in the solution space of the system.

The suggestion that Poligon is a blackboard system is a little controversial. There are a number of respects in which this is not a satisfactory label. This term will, however, be used freely from now on for lack of a better label. The reader is encouraged to substitute for the term "Blackboard system" any term, such as "Frame System" which seems best to fit his mental model of what is being described.

1.2. Poligon's Model of Parallelism

It seems appropriate here to describe Poligon's model of parallelism. In its simplest form this can be thought of as *An Element in the Solution Space as a Processor.*

This gives some idea of the granularity that is being sought. It is, however, by no means the most efficient way to implement Poligon. Poligon programs want to be able to execute rules and parts of rules associated with a particular *Node* in the solution space in parallel. These rule activations need processors, on which to execute.

Thus a modified version of Poligon's model of parallelism could be *A Rule Activation as a Process, with sufficient processors to cope with the parallelism exhibited by the rule during its activation.* This tends towards a mapping of solution space elements onto a cluster of processors to service the rule activations. In practice, however, a number of nodes might be folded over the same set of processors, either because nodes become quiescent or because the load balancing in the system is sub-optimal.

2. Related Work

Work in this field falls into two distinct categories; work on parallel knowledge based systems and work on languages for parallel symbolic computation. The former is, at present, a very sparse field and, will not be discussed here, though some references are given in § 6. The latter is much more highly developed.

Much work is already being done on parallel languages for general computation. Amongst these languages are **Actors, MultiLisp** and **QLisp** on the one hand and concurrent logic programming languages and purely functional languages on the other. Often missing from this

work is a thrust toward the investigation of large applications in parallel domains, for instance the development of parallel knowledge representation and problem solving systems. This is, of course, what Poligon attempts to do. This section will discuss briefly Actors, QLisp and Multilisp, since these are the parallel symbolic computation languages which are most relevant to the development of Poligon and the software which lies beneath it.

2.1. Actors

Actors [Hewitt 73] probably come the closest in their behaviour to Poligon, at least at an implementation level. Actors are independent, asynchronously communicating objects. As is the way with purely object oriented systems they communicate only through message passing and have tightly defined operations. The mutual control of Actors an parallelism is achieved by the support of procedure call and coroutine model message passing. The modularity afforded by this sort of programming metaphor may well be especially useful for the programming of distributed-memory, message-passing hardware, since having a close match between the hardware and software metaphors is likely to achieve better performance. It is not in any way surprising that the operating system level software, which underlies Poligon, is founded on many of the same principles as Actors. It has yet to be seen whether this programming methodology is able in practice to extract significant amount of parallelism from problems, though clearly this project hopes that it is.

2.2. MultiLisp and QLisp

MultiLisp [Halstead 84] and QLisp [Gabriel 84] are lumped together because, at least in some senses, they have strong generic resemblances. They are both, at the user level, extensions to existing Lisp dialects which provide mechanisms for the expression of parallelism, such as parallel Let constructs and parallel function argument evaluation (QLet and PCall). It is assumed by both of these systems that the hardware at which they are targeted is a form of shared-memory multiprocessor. Although there is no particular reason why such systems could not be implemented on a distributed-memory system, they are optimised for shared-memory multiprocessors. These are currently the most readily available form of multiprocessor. They would, however, need significant extensions in order to be able to exploit a distributed-memory system as is shown in CAREL [Davies 86], an implementation of QLisp for distributed-memory machines. The assumption of shared-memory, MIMD processors in these systems imposes constraints on the languages. They assume, at least to an extent, that processes will be expensive and that the user must have control over their creation. Poligon assumes quite the opposite.

3. The Design of Poligon

Poligon will be discussed first in terms of the way in which the language relates to the problems being solved and its underlying systems. Next the language will be discussed in terms of the requirements for languages in general and parallel languages in particular.

3.1. Background and Motivation

The philosophy behind the design of Poligon comes from intellectual and pragmatic pressures. It attempts to steer a middle course between the extreme purism of applicativists and the extreme pragmatism of the proponents of side-effects.

From the outset, the project was oriented towards real-time problem solving. Blackboard systems are well known to be of interest as tools in the knowledge engineer's toolkit. Little work has been done to investigate the appropriateness of the blackboard metaphor to parallel execution or the meaning of parallel blackboard systems, though it is frequently claimed that they are full of latent parallelism. The excellent formal properties of pure applicative and logic languages may well be of little use in a system which, for whatever reasons, needs to express side-effects and which has to cope with real-time constraints. Poligon is a system in which

some of the formal rigour of truly applicative systems has been put aside in favour of a pragmatic approach to the exploitation of parallelism.

The **BB1** project [Hayes-Roth 85], also a project at the HPP, is an attempt to investigate the behaviour of highly controlled problem solving systems. It attempts to use a great deal of meta-knowledge and makes significant use of globality of reference in order to support an holistic view of its solution space, thus providing a basis for meta-level reasoning. The Poligon project is an attempt to investigate quite the reverse. Poligon has very little support for meta-knowledge and allows no global data or global view of the solution space whatsoever. The purpose of this experiment is to determine whether a system, unconstrained by a great deal of serialising control knowledge, might still be able to find useful answers faster than an highly controlled system, such as BB1, which would be extremely difficult to speed up significantly through parallelism.

The Poligon system pictures the elements in its solution space as processes resident on processors distributed across a grid, with the code necessary for them intimately associated with them. Because no global control is permitted in Poligon the activation of rules is necessarily completely daemon-driven.

The project hopes to achieve significant speed-up through parallelism. This can be done only if much parallelism is extracted from the problem. Ideally, the system would try to achieve its parallelism by exploiting parallelism in the program's implementation at a very fine grain. This can, in principle, extract the maximum amount of parallelism available. On its own it has drawbacks, however. The costs of processes and the problems of synchronisation at a fine grain size make it difficult to exploit such parallelism without the use of hardware mechanisms significantly different from those available with prevailing technologies. This approach is also only part of the story. It neglects the fact that a properly parallel decomposition of the source problem is crucial to finding a lot of parallelism. One could summarise the problems, therefore, as expressing the problem in a sufficiently parallel fashion and the matching of the parallelism in the program to the grain size of the underlying hardware. Poligon addresses these issues.

Parallelism is very hard to find in conventional programs. Applicative systems have an advantage in this respect because of their relative lack of need to express parallelism explicitly. Their unchanging semantics when parallelism is introduced eases matters considerably. Poligon has attempted to learn from this and has pure applicative semantics in a number of areas but takes a different approach to the finding of parallelism in programs. It attempts to execute everything in parallel that it can and leaves it to the programmer to find any serial dependencies.

When the parallelism in a program is user-defined, problems can result from an inappropriate match between the granularity of the parallelism expressed in the program and the granularity of the underlying machine. In systems of the size and complexity of a typical Poligon application such a match would be particularly difficult to find because of the large number of processors involved and because it would be difficult for the user to keep track of the location of his data in the processor array. These characteristics are a consequence of the highly variable and data dependent state of the solution space in such programs. Poligon, because of its structure, should be able largely to obviate such granularity mismatches because parallelism is defined and controlled by the system and the Poligon system is closely matched to the granularity of the underlying system.

It is often thought that problems suitable for solution by means of the blackboard model tend to partition their solution spaces into what look rather like pipe-lines. Pipe-lines are, of course a well known form of parallelism. In practice pipes in such systems are not pipes in the normal sense, since they are more like "leaky" pipes. It is one of the prime objectives of these systems to reduce the amount of data as it percolates up through the abstraction hierarchy of the solution space. Because of the reduction in the data rate flowing in these pipes the contention problems that one might expect when pipes are connected into trees, as they often are, are alleviated.

A significant limitation of the performance of pipelines is that, at best, the parallelism that they can produce is proportional to the length of the pipe. This would typically be only of the order of half a dozen sections. This is clearly not the "orders of magnitude" of performance improvement that we all hope for. In practice, though, given a large enough problem, it is often possible to set up a large number of these pipes side-by-side. It is one of the major objectives of the Poligon language to encourage, facilitate and reward the decomposition of problems so that this form of independence can be exploited, so that such pipes will be created by the system.

3.2. Language Requirements

Poligon is a language which is by no means directed at general computation. It is nevertheless intended to be used for the solution of large, complex problems on distributed-memory parallel hardware. The following is a brief list of the ways in which Poligon attempts to address some of the primary requirements of programming languages.

- The language should provide a tangible method of expressing the ideas of the programmer.

  The Poligon language has been written with considerable input from those with experience in problem solving systems in the application domains at which it is targeted. It is therefore intended to match the ideas of the "Expert", whose knowledge is to be encoded, but in a domain independent way.

- The compiler[2] should provide a mapping between the language and the underlying systems, be they hardware or software.

  Poligon's compiler compiles Poligon language source into code understood by the underlying *Lisp* system and the concurrent object-oriented operating system running on its target hardware.

- The language should abstract the programmer from its underlying systems.

  The Poligon system shields the user from all aspects of the underlying hardware such as the topology of the processor network, the message-passing behaviour of the hardware and the location of any code or data within the network.

- The language should provide mechanisms for the exploitation of the underlying systems to good effect.

  The underlying hardware and software systems are exploited in a number of ways in Poligon. Firstly the language encourages the user naturally to decompose his problem into a form which will map efficiently onto the underlying hardware. Secondly the language offers a number of application-independent, high-level constructs, which are designed to exploit the hardware to the full. These topics are covered more fully in § 4.

- The language should allow the development of software faster than would be the case if it were to be developed in a less abstract form.

  Considerable effort has been spent on making the Poligon language a high level way to describe the solutions to parallel knowledge based system problems. A high level language with such features as infix, user-definable operators and user definable syntax, provides a natural way for the expert to implement his knowledge.

  Much effort has been spent also on integrating the Poligon system cleanly into the program support environment of the Lisp Machines on which it runs. For instance, incremental compilation is supported from within the editor.

---

[2]The term *Compiler* is used in its most general sense here, perhaps an interpreter or a machine which is clever enough to execute the language specified directly.

- The language should assist the development of reliable, maintainable and modular software.

  Language features are provided to minimise the possibility of inconsistent modifications to the source code and the structure of the language and its semantics are defined in a manner which minimises the probability of complex bugs being introduced by asynchronous side-effects.

  A sophisticated set of debugging facilities is provided. A system that emulates the semantics of full, parallel Poligon programs as closely as possible in a serial environment has been produced. The user is able to debug his program serially to remove all possible serial bugs and bugs due to the non-deterministic execution order of Poligon programs before it is ported to the full parallel environment.

In addition to these requirements a language targeted at parallel hardware should have a number of attributes which reflect the parallel nature of the target hardware.

- The language should address the granularity of the hardware.

  Poligon is closely matched to the granularity of the hardware at which it is targeted. It is generally expected that the solution space of the problems addressed by Poligon programs will have of the order of thousands of nodes. This is of the same order as the granularity of the hardware.

- The language should provide a mechanism for the extraction of parallelism from programs and from the programmer.

  Poligon extracts parallelism from programs and the programmer in two main ways. First the decomposition of the problem is encouraged to be as modular as possible. Secondly the semantics of Poligon programs are such that almost all of the program can be executed in parallel without changing their behaviour from that seen during serial execution. This allows the system to execute most operations in parallel if it has the resources to do so.

- The language should, where appropriate, shield the programmer from those details of the hardware which are particular to parallel computing engines, such as topology.

  The hardware, on which Poligon programs runs, causes Poligon programs to have to cope with communication between solution space elements on different processor sites. All such message passing is hidden from the user. In fact the Poligon language has no concept of message-passing at all.

  Futures are used for all remote operations in the user's program. The hardware implements these such that there is no efficiency penalty associated with creating futures for such remote accesses. The Poligon language copes with these invisibly to the programmer.

As can be seen quite easily from the above one of the factors that must be well understood before a language is designed is the general purpose of the language and the level of generality that is expected of programs written in it. A language, whose sole purpose is the expression of solutions to huge matrix problems on systolic hardware might well be justified in expecting the programmer to express, at quite a low level, the mapping of the program onto the hardware provided. This is less likely to be a reasonable expectation of a language targeted at the solution of large, complex problems of an unpredicatable, dynamically-varying or data-dependent nature. Poligon is a fairly general purpose programming language with a very definite bias.

4. Abstractions in Poligon

To cope with Poligon's view of parallelism and with the chaotic execution of rules (see § 1) a number of linguistic abstractions are provided.

Poligon provides abstractions for knowledge representation, control, data, parallelising, real-time and side-effect control. These will be described briefly in this section.

4.1. Knowledge Representation

Knowledge is traditionally represented in blackboard systems in a number of ways, listed below.

- *Declarative Knowledge* is encoded in *Rules*.
- *Procedural Knowledge* is encoded in procedures.
- Knowledge concerning the sequencing of activities is encoded in the scheduling mechanism.
- Knowledge about the structure of the solution space is encoded by the definition of the structure of the blackboard.
- Knowledge about relationships between the objects in the system is often encoded using a Link mechanism.

These all represent knowledge about the application domain. In addition, there is in any program a large body of implicit knowledge concerning the semantics of assignment, sequencing and the system's function as a whole, especially in for systems with poor formal properties. This will not be discussed here. The Poligon language does, however, go to considerable effort to make the semantics of the Poligon system as clear as possible.

4.1.1. Declarative Knowledge

The encoding of *Declarative Knowledge* in blackboard systems is conventionally done in *Rules*[3], which exist within scheduling units known as *Knowledge Sources*. Poligon also has the concept of Rules and Knowledge Sources, though their meaning is somewhat different. Unlike serial blackboard systems, the rules in a Poligon system are activated autonomously and asynchronously.

Existing blackboard systems usually suffer from a confusion and overloading in the semantics and purpose of knowledge sources. It is useful to collect one's knowledge of one subject together into one chunk. These chunks are knowledge sources. Sadly, the implementors of blackboard system frameworks often think of knowledge sources as scheduling units and thus design their scheduling strategies around the idea of the "invocation of knowledge sources", even though it is by no means necessarily the case that it is appropriate to schedule all of knowledge in a chunk at the same time. This has a detrimental effect on the modularity of the system.

In Poligon, knowledge sources are used as linguistic and software engineering abstractions provided for the programmer in order to allow him to collect related knowledge together. There are no scheduling semantics associated with knowledge sources in Poligon. Because of the underlying system's daemon-like rule triggering mechanism the rule writer is allowed completely to decouple the concept of *scheduling* from the concept of *chunks of knowledge*.

Rules are activated as a result of "events" happening to the fields of nodes (see § 4.3.1). These events can be caused either by a write operation to a field, by a semaphore being waved at a field or by the real-time clock.

A powerful *Expectation* mechanism is provided, which allows the dynamic placement and specialisation of rules. An Expectation is a way of expressing model-based knowledge. Given

---

[3]The term *Rule* is used here in the sense of "Pattern/Action pairs". It should be noted that these are quite unlike the structures called rules used, for instance, in **Prolog**. Pattern/Action rules move towards a solution to their problem by performing side-effects on their environment, in this case the blackboard, not through unification.

a particular model of the behaviour of a system, certain changes might be expected if the model's interpretation of the world is correct. Expectations allow such changes to be watched and even allow their associated rules to be triggered if the changes do not happen in a given time. Such expectations can be placed to watch for events happening, or not happening, in specific places on the blackboard, at specific times. Expectations provide a focussing mechanism[4] and, coupled with the system's ability to trigger[5] rules and "time-out" unsatisfied Expectations on the basis of the real-time clock, Poligon allows complex time-critical knowledge to be expressed and applied simply.

An example rule is shown in figure 4-1.

4.1.2. Procedural Knowledge

Procedural Knowledge is an all encompassing term usually used indiscriminately to describe both knowledge about the relationships between values (*Functions*) and the mechanisms for performing side-effects and for sequencing events (*Procedures*). This is often a result of such systems being built on top of Lisp systems, which fail to draw distinctions between procedures with side-effects and those without. Poligon does not allow the encoding of arbitrary knowledge into procedures. Only side-effect free functions are allowed. Side-effects are permitted only in the bodies of rules, where they can be controlled.

4.1.3. The Sequencing of Activities

In most blackboard systems knowledge of the required sequencing of events at a macroscopic level is expressed by the implementation of the system's scheduler. In many cases, such as AGE [Nii 79] this scheduler has fixed characteristics and the application has a fixed interface to it. In others, such as MXA [Rice 84], the user can specify the characteristics of the scheduling of knowledge sources. Poligon provides no such mechanism. Since all rules are activated as daemons, entirely asynchronously, the only analogue of scheduling is the implicit sequencing of the activation of rules due to some rules causing changes that trigger other's rules.

4.1.4. The Structure of the Solution Space

Poligon is unlike most blackboard systems in this respect. Most blackboard systems partition the blackboard into *Levels*, which represent the hierarchy of abstraction in the solution space. Poligon uses a much more general representation which is like that of some *Frame* systems, providing a "Class" mechanism with user defined classes and metaclasses, and compile-time and run-time inheritance. The functionality of the class mechanism in Poligon is a superset of that of the levels provided by most blackboard systems. The programmer can, of course, represent his solution simply using classes as levels in Poligon if he wishes. Classes are discussed more in § 4.3.1.

4.1.5. Knowledge about Relationships

Relationships between entities in blackboard systems are often expressed by a form of *Link* mechanism. Sometimes this link is not so much a part of the system as a reflection of the fact that fields in nodes can have as their values other nodes in the system. Other systems have more sophisticated mechanisms that express links explicitly and allow property inheritance along links, e.g. BB1, or the propagation of likelihood, e.g. MXA.

Poligon has a number of system defined relationships; "Is an Instance of", "Is a part of" and "Is a subclass of". The user can define arbitrary relationships between nodes on the blackboard. These links allow property inheritance and are, themselves, represented as nodes and so

---

[4] It should be noted that the term *Focussing mechanism* is used in a more general sense than by many blackboard systems. There can be any number of such foci all acting in parallel in a Poligon program. The expectation mechanism is another way of applying knowledge in order to take advantage of some local circumstances in order to solve a problem more efficiently or cleanly.

[5] A rule is said to have been *Triggered* when it is activated so that it tries to evaluate its preconditions and body.

The following is a trivial example rule, which shows a small set of the features of Poligon. This rule could be interpreted as saying; *"If the most recent two phonemes that have been seen are "oo" and "ph" then the word is "foo".* Having concluded this the rule finds the set of sentence components, which represent potential conclusions of the word "foo", and sets them so that they are no longer marked as hypothetical. It also makes a *Sentence-Component* type node, which represents the word "foo", which has been found.

```
Rule : Find-the-word-Foo
    Class : Phoneme
        { Class of nodes with which the rule will be associated }
    Field : uncorrelated-phonemes
        { Try to activate this rule when this field is changed }

Definitions :
    all-phonemes-in-order ≡ The-Phoneme⊕↑uncorrelated-phonemes
        { The operator "⊕↑" returns all values in a field in }
        { time order.  The-Phoneme represents the node, that }
        { triggered this rule }
    most-recent-phoneme ≡ all-phonemes-in-order·Head
    next-most-recent-phoneme ≡ all-phonemes-in-order·Tail·Head
        { Head and Tail are like CAR and CDR only they operate }
        { on lists, Lazy lists and Bags }

Condition Part :
    When : all-phonemes-in-order·length-of-list ≥ 2
        { The "When" part is a locally evaluable precondition }
    If   : most-recent-phoneme·Sound = "oo"
           And next-most-recent-phoneme·Sound = "ph"
        { The precondition for the Rule }

Action Part :
    Definitions :
        new-sentence-component ≡ New Instance of Sentence-Component
        { The creation of the new Sentence-Component node }
        hypothetical-foos ≡
        { A Bag of words, which are "foo" }
            Subset of Words which satisfies
                λ(a-word)
                    a-word·hypothetised And a-word·letters = [ f o o ]
                Endλ

        { Process all elements in the Bag hypothetical-foos }
    Changes :
        In Parallel for each a-word in hypothetical-foos
            Change Type    : Update
            Updated Node   : a-word
            Updated Fields : hypothetised ← nil

        { Set fields of new sentence component in }
        { parallel with updating the elements in the Bag }
    Changes :
        Change Type    : Update
        Updated Node   : new-sentence-component
        Updated Fields : letters ← [ f o o ]
                         constituents ← List(next-most-recent-phoneme,
                                             most-recent-phoneme)
```

All of the actions taken by this rule are performed in parallel, since they are independent of one another, though there is, of course, a serial dependency between the condition part and the action part of the rule.

Figure 4-1: An example Poligon rule

can have attributes in the same way that any other nodes can. Links are therefore first-class citizens in Poligon and they allow Poligon programs to act like semantic nets.

4.2. Control Abstractions

The flow of control is a rather evanescent concept in a Poligon program. Any rule can be triggered at any time. It is important not to think of the control flow in a Poligon program in the same terms as that of a conventional serial program. There is a well defined flow of control within rules; the action part of a rule is activated after the condition part, upon which it is predicated. Apart from this, however, there is no flow of control in any normal sense. It should be noted also that what little flow of control there is only specifies the strict ordering of activities. The execution of a sequence of actions can be interrupted at any time. The size of the atoms for Poligon's atomic actions is very small.

The triggering of rules is controlled by the user associating rules with particular fields of nodes or classes of nodes on the blackboard. The triggering of rules occurs when a field, which is being watched in such a manner, is updated or is semaphored. A semaphore mechanism is provided to allow rules to be triggered without a field being updated. This provides a form of explicit event-based programming, if it is needed.

Clearly one of the objectives of the design of the Poligon language is to provide a language in which it is simple to express logically distinct pieces of knowledge, independent of other such pieces of knowledge. The decomposition of the problem in this manner causes the system to appear to iterate towards the solution of its problem by small, simple and discrete steps, rather than by complex, giant leaps.

4.3. Data Abstractions

Poligon provides a number of distinct data abstractions. One is characteristic of other blackboard systems, one of pure functional languages and one is rather novel.

- The structure of the blackboard is characterised by being made of *Nodes*, elements in the solution space. These have a user-defined, record-like structure.
- Lazy evaluation is supported.
- Bags are supported as data structures, which parallelism enhancing.

Numerous operations are defined for these data abstractions, particularly a number of generic operations which can be applied to lists, lazy lists and bags, which shield the user from the underlying data structures used by the system or by other segments of his program.

4.3.1. The Structure of the Solution Space

The most obvious data abstraction provided by Poligon is similar to that provided by conventional blackboard systems, that is, the *Node* on the blackboard as an element in the solution space. Such nodes are record-like internally. They have named fields, which can often contain multiple values to be associated with that name. Poligon provides this but also goes beyond it.

Conventional blackboard systems, such as AGE, tend to provide nodes on a blackboard divided into groups, often called "Levels". "Levels" themselves are not represented. Arbitrary use of global data, held in global variables, distinct from the blackboard is also allowed.

Poligon has a much more regular representation for data. The nodes are represented as instances of *Classes*. The Classes themselves are represented as Nodes, which "control" their instances. Knowledge concerned with classes as a whole can be associated with these nodes. Shared, global variables are not allowed in Poligon.

Poligon also provides;

Superclasses Classes that provide characteristics to the instances of classes. These can be thought of as templates for the instances.

Metaclasses Classes that provide characteristics to the classes themselves. These can be

thought of as templates for the classes.

Thus the classes are themselves instances of metaclasses, which can be user defined, such that instances of a given class can have any number of superclasses, i.e. component templates, and any number of metaclasses, i.e. component templates for their parent class. It is possible to instantiate classes any number of times, as well as their instances.

Automatic property inheritance allows shared data to be located on locally central nodes, which are immediately visible to the interested parties. This distributes shared data in such a manner as will, hopefully, minimise hot-spotting.

An example class declaration, the specification of a template for a class of nodes, is shown below. The declaration defines a class of nodes called *Words*, each instance of which has two fields (slots) called *Letters* and *Sound*.

```
Class Words :
    Fields :
        Letters
        Sound
```

Extensions to this sort of syntax allows the definition of superclasses and metaclasses within class declarations. The following example defines the class *Sheep*. Each instance of the class *Sheep* will have the characteristics defined for sheep and for mammals. The class called *Sheep* (an instance, in fact, of the class *Meta-Sheep*) has the characteristics of *types of animals*.

```
Class Types-of-animals :
    Fields :
        Rate-Of-Breeding

Class Mammals :
    Fields :
        Colour-of-fur
        Number-of-legs : 4

Class Sheep :
    Metaclasses  : Types-of-animals
    Superclasses : Mammals
    Fields :
        Thickness-of-wool
        Flock
```

4.3.2. Lazy Evaluation

Lazy Evaluation is supported in the guise of *Lazy Lists*, *Lazy Function Arguments* and in the form of the lazy association of expressions with names. The following is an example of the lazy association of a name with a value. The name *A-Meaningful-Name* is associated with the value of the call to the function *An-Expensive-Function*[6].

```
Definitions :
    A-Meaningful-Name ≡
        An-Expensive-Function(an-arg, another-arg)
```

The value of an item defined in a *Definitions* construct is always a future if it is possible to evaluate it as a future.

---

[6]Suitable *Force* operations are provided so that the time of evaluation can be controlled by the program if necessary. These force operators allow the program to perform *Eager Evaluation* if it is needed.

4.3.3. Bags

One abstraction suited particularly to the parallel mode of execution of Poligon programs is the *Bag* data type. Bags are implemented in Poligon so that they are formed as the result of efficient parallel operations and can be processed in parallel efficiently. Even when the elements of Bags are processed serially they perform efficiently. The lack of a defined ordering in the Bag means that the system can always return the first satisfied Future out of a Bag of Futures, causing minimum waiting for values. Similarly, when a program attempts to extract an element from a bag and there are no satisfied elements the process in which this happens will go to sleep until the next available future is satisfied.

A Bag is generated, for instance, as the value of the following expression. It is a Bag, which contains all of the *Words*, whose *Sound* is *"phoo"*[7].

```
Subset of Words For Which Element · Sound = "phoo"
```

4.4. Parallelising Abstractions

Poligon supports data representations which are designed to give the user a high level handle on the exploitation of parallelism. Most values computed in Poligon are derived as *Futures*. Computation is decoupled from the expressions which reference values. Futures are, however, completely invisible to the user in Poligon. It understands which functions are strict in their arguments and so waits for the satisfaction of a Future only when it is required. The programmer can, of course, declare his own non-strict functions and operators. All *DeFuturing* coercions are performed automatically by the Poligon system. Thus the following expression will deliver a list with two elements, one of which is the value of *a* and one of which is the sum of *b* and *c*. The first will be a future, if *a* is. The second will be the DeFutured value *b+c*.

```
List(a, b+c)
```

The efficient use of the bandwidth of the processor interconnection network is enhanced by the use of *Broadcast* and *Multicast* operations. Broadcast messages allow messages to be sent to every node in the system in a single operation. Multicast messages allow messages to be sent to a collection of nodes in a single operation. The Poligon system uses these extensively in the processing of the Bag data type and in the execution of groups of actions in parallel. It uses the same mechanisms to provide an efficient implementation for searching a collection of nodes on the blackboard for patterns, which tends to cause significant slowing of serial implementations because of the combinatorial nature of such searches. It allows the blackboard to be searched for bags of matching nodes in a single, fast operation. This provides a significant improvement over the serial construction of such collections.

4.5. Real-time processing

Real-time processing brings its own problems. Poligon provides a simple and regular mechanism for defining the interface between the Poligon system and its signal data. This data can be from an arbitrary number of different types of sources and is posted on the blackboard asynchronously.

Poligon also provides a mechanism by which each datum is timestamped from the time that it enters the system. These timestamps are propagated automatically by the system so that it is trivial for the programmer to manipulate time-ordered collections of values. This mechanism is required because the conventional implicit time ordering of data in lists cannot apply here

---

[7]The expression "Element · Sound" denotes extracting one of the values associated with the "Sound" field of the potential element in the bag. "·" is an operator that selects which of the values associated with the field is to be delivered.

and the non-ordered nature of Bags is sometimes not sufficient.

4.6. The control of assignment

Assignment is something which is likely to cause significant problems in any parallel system. Poligon constrains assignment in a number of ways. Side-effects are only permitted on the fields of nodes. All side-effects can be monitored by rules that might be interested in the changes to values. This removes the possibility of the knowledge base getting confused because of surgical side-effects to data structures at arbitrary times and at arbitrary places in the processor network. Assignment is also constrained so that all of the updates to the fields of a given node are done atomically, before any rules which might be triggered by these changes are allowed to trigger. Such atomicity helps to preserve the consistency of the system.

An example of a collection of updates to fields of a given node is given below. In this example the node *an-instance-of-words* is having two of its fields updated; *Sound* and *Letters*. Operators, such as "←", allow different sorts of modifications to be made to fields. Such operations might be "add this value to the values in this field" or "replace all of the values in the field". This avoids complex and potentially expensive expressions in the old value of the field being evaluated non-locally.

```
Change Type    :  Update
Updated Node   :  an-instance-of-words
Updated Fields :  Sound   ← "phoo"
                  Letters ← [ f o o ]
```

5. Conclusions

This paper has described Poligon, a language and system for the investigation of problem solving on distributed-memory, parallel hardware. The language was described in the context of related work in the field and in terms of the abstraction mechanisms provided. No significant description of the underlying run-time support has been given.

The Poligon system is still young. Only recently have applications been mounted on it in earnest. Two distinct applications in the field of real-time signal processing are now being implemented and more applications are likely to be started in the near future. Poligon has proved to be well suited to these applications as far as they have gone. No results from the simulation process regarding the performance of Poligon programs are yet available. Significant problems have been found in the simulation of the fine-grained parallelism required by the Poligon metaphor. Such simulations are very time consuming, prone to bugs in the underlying system software and simulator, and are difficult to debug. It is for these reasons that Poligon also has a serial version, Oligon, which accurately emulates the behaviour of the parallel system but without true parallelism. A simulated processor array of 256 processors has recently been made available to the users of Poligon. This simulation will allow more satisfactory investigation of the properties of Poligon programs in the future.

6. Further Reading

For a significantly more detailed treatment of the Poligon language and system the reader is encouraged to consult [Rice 86].

The following topics were not described or discussed but are relevant to the work described above. The reader is encouraged to consult the following for further information;

- [KSL 85] for a description of the Advanced Architectures Project of which Poligon is a part.
- [Delagi 86] for a description of CARE, the hardware simulator used by Poligon, and of the particular hardware being simulated.

- [Schoen 86] for a description of CAOS, the concurrent object oriented system running on the CARE machine, which Poligon uses as its operating system.
- [Ensor 85], [Lesser 83], [Aiello 86] and [Fennel 77] for other approaches to parallel problem solving using blackboard systems.

References

[Aiello 86] Aiello, Nelleke.
The Cage User's Manual.
Technical Report KSL-86-23, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.

[Davies 86] Davies, Byron.
Carel: A Visible Distributed Lisp.
Technical Report KSL-86-??, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.

[Davis 77] Davis, R. and J. King.
An Overview of Production Systems.
In E.W. Elcock and D. Michie (editor), *Machine Intelligence 8: Machine Representation of Knowledge*, . John Wiley, New York, 1977.

[Delagi 86] Bruce Delagi.
CARE User's Manual
Heuristic Programming Project, Stanford University, Stanford, Ca. 94305, 1986.

[Ensor 85] Ensor, J. Robert and Gabbe, John D.
Transactional Blackboards.
Proc. of IJCAI 85 :340 - 344, 1985.

[Fennel 77] Fennel, R. D. and Lesser, V. R.
Parallelism in AI problem solving: a case study of Hearsay-II.
IEEE Trans on Computers, C-26 :98-111, 1977.

[Gabriel 84] Gabriel, Richard P. and McCarthy, John.
Queue-based Multi-processing Lisp.
Proceedings of the ACM Symposium on Lisp and Functional programming :25 - 44, August, 1984.

[Halstead 84] Halstead, Robert H. Jr.
Implementation of Multilisp: Lisp on a Multiprocessor.
Proceedings of the ACM Symposium on Lisp and Functional programming :9 - 17, August, 1984.

[Hayes-Roth 85] Barbara Hayes-Roth.
Blackboard Architecture for Control.
Journal of Artificial Intelligence 26:251 - 321, 1985.

[Hewitt 73] Hewitt, C., P. Bishop, and R. Steiger.
A Universal, Modular Actor Formalism for Artificial Intelligence.
Proceedings of IJCAI-73 :235 - 245, 1973.

[KSL 85] Knowledge Systems Laboratory.
Knowledge Systems Laboratory 85, incorporating the Heuristic Programming Project.
KSL, Dept of Computer Science, Stanford University, 1985.

[Lesser 83] Lesser, Victor R. and Daniel D. Corkill.
The Distributed Vehicle Monitoring Testbed: A Tool for Investigation Distributed Problem Solving Networks.
The AI Magazine Fall:15 - 33, 1983.

[Nii 79] Nii, H. P. and N. Aiello.
AGE: A Knowledge-based Program for Building Knowledge-based Programs.
Proc. of IJCAI 6 :645 - 655, 1979.

[Nii 86] Nii, H. P.
Blackboard Systems.
AI Magazine 7:2, 1986.

[Rice 84] Rice, J. P.
The MXA user's and writer's companion
Systems Programming Ltd, The Charter, Abingdon, Oxon, UK, 1984.

[Rice 86] Rice, J. P.
The Poligon User's Manual.
Technical Report KSL-86-10, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.

[Schoen 86] Schoen, Eric.
The CAOS System.
Technical Report KSL-86-22, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.